NaturalLanguageProcessing-Lecture07

Instructor (Christopher Manning):Okay, hello again. I'll get started. So the topic that I want to talk about today is how to use classification methods, and NOP in particular ought to start looking at the kind of discriminative classifiers that are being used a lot in recent NOP and, in particular, in that space what NOP people call maxent classifiers. And I'll eventually get around to explaining the reason for that name, but they may be more familiar to you as the space of logistic regression exponential model style discriminative classifiers.

Before I get started on that, since I was in a little bit of a rush right at the end of last time, I just wanted to quickly re-orient us and think about this change of address e-mail problem. And, in particular, I think it kind of nicely illustrates what you can do with simple classifiers with just assigned things to one of a fixed number of classes, versus what motivates the kind of more complicated sequence models that are used a lot of the time in NOP.

Okay, so the problem here was what the Kushmerican co wanted to do was get out e-mail change of addresses so they could get 'here is a new e-mail for a person.' And the way they set it up was to say, well, we can do this as simply two binary classifiers. And the first binary classifier just looks at the whole e-mail and says, “Is this a change of address e-mail or not?” So that's just a two-way choice, and the way it does that, in the terms that we started this course with, is you take a bunch of change of address e-mails, and you build a language model from them.

It could be your favorite kind of language model. It can be smoothed in your favorite way you build a language model. You then take a whole bunch of non-change of address e-mails, and you build a language model from them. And so then you can ask which of those two language models gives higher likelihood to an e-mail. Is it generated with higher likelihood with a change of address language model or the other e-mail language model? And then if you add to that only one more factor, which is a prior probability estimate, an estimate of what percentage change of address e-mails, and you multiply in that waiting factor from the prior probability of having a change of address e-mail.

What you get out is normally called a multi-nomial naive based classifier. The multinomial naive based classifier is exactly a class conditional language model classifier. So these things kind of do connect together. So then once you've made that first decision, the second decision is to say that in most e-mails there are actually multiple addresses mentioned, and so which one is actually the new address of the person?

And the way that that is handled – and this has been a very common technique that has been used for a lot of problems in NOP, is that you effectively define a mini-document. And your mini-document is defined as – I think it's six, or maybe it's eight. It's about six tokens to the left of the thing that you're interested in classifying, and six tokens to the right. So you have to find this little window, which is a mini-document. So then you ask this mini-document, “Does this mini-document look like new e-mail address?”

Binary classification, again, two ways. Does this e-mail look like new e-mail address through binary classification? And if you're lucky, you decide this one is the new e-mail address, and then you've got out the right result, which is what's shown down here. And so these people made this system. They did a couple of different versions depending on whether the features were just words or made use of phrases. So in the better of the two systems, the [inaudible] classification was extremely high in deciding whether address was a new address was pretty high so they could this task very well.

Somebody else asked, afterwards, why do you need two classifiers? Why can't you just have one classifier that just looks at every e-mail address and does the second classification task? And I think the answer to that is shown here that making the overall decision on the whole e-mail is much easier. The biggest danger in this task is that you get killed by false positives since most messages aren't change of address messages. So you want to have as high an accuracy as possible at filtering out things that aren't change of address e-mails, and that gives you a fairly good take for doing that.

This looks really easy. All you've done is build two binary classifiers, and you've got an information extraction system ready to go. The crucial thing to note, though, is what made it so easy. What made it so easy was that candidate e-mail addresses deterministically identifiable by writing a regular expression for e-mail addresses. So if you can write a regular expression that produces candidates, then if it's almost 100 percent accurate, you can just stop.

If it's not quite that easy a problem, and you have to look in the context to see if it's the right one, whether it's the right e-mail address – or in the earlier example, whether it's the right price for the product that you're shipping to the US. Then you can use a probabilistic model as a simple classifier where your probabilistic model just looks in the environment of candidates and makes a decision whether this is a good one or not.

A lot of the time things aren't that easy for what we want to do. The kind of examples that aren't that easy are the kind of examples I showed earlier in the last class when we were doing named entity recognition, and we were looking for person names, company names and things like that. We had that example where it was First National Bank of Ottawa or something like that. There are all kinds of possibilities there whether it should be Bank of Ottawa, or First National Bank of Ottawa should be recognized as the company name.

When you're in that situation, you want to be comparing all of those possibilities off against each other. You need something that is considering different sub-sequences and their relative likelihood. You can't do that with a simple classifier.

Okay, is everyone good to there? Okay, so what I then want to go onto today is start introducing discriminative classifiers, and then lead from there into talking about how discriminative classifiers are used in the context of sequence models. So using this kind of discriminative classifier will be half of what you do in the third assignment. So we're kind of, one-by-one, working through all the assignment topics. It's also – I think these slides give a fairly readable introduction by themselves, but there's also a couple of sections of Jurafsky and Martin which talk about using maximum entropy mark-off models, which is the technology that we're building to here.

So the primary contrast in just sort of scene setting as to how this fits together – the things that we've started off looking at of what get called generative probabilistic models. So our language models that we did first, our IBM machine translation alignment models, or the naive based classify that I briefly mentioned. They're all generative models, and generative models are these models – I'll give more details later. You kind of write this sort of probabilistic story about how the data is generated, and you multiply conditional probabilities together.

That kind of used to be the sort of state-of-the-art that everyone used for all NOP problems if they're doing probabilistic NOP in the 1990's. But really, in this decade, a huge amount of the work that is being done has switched to using discriminative models, so I want to spend a bit of time talking about discriminative models, and how they work, and how people build them. This tries to be a little summary of why people like discriminative models. They work well. You get high accuracy. That's a reason to like them.

Why do they work well? I'll explain in more detail later. The biggest reason why they're powerful and useful in NOP problems is that discriminative models, in some sense, are friendly to a non-techie linguistically oriented data-analysis person. They actually provide a framework in which it's very easy to encode different sorts of linguistic knowledge and have someone else's clever package do the tricky number optimization stuff for you.

So although there's a certain amount to understand on how they work, at the end of the day they provide a sort of very friendly abstraction where it's easy to go about building these models without really understanding the details of how they work, providing you actually understand some facts about the domain, and how to define features.

Joint versus discriminative models. So joint models, which are generative models – but ultimately what they're doing is that they're modeling this joint probability, so the joint probability of their data that you've observed, and the stuff that you didn't observe, which is here. I'm calling classes with a C, but that's also like the alignments when we're doing MT. So those are the kind of examples we've seen. This was where we always told the noisy channel models story, which was base of all where we're doing a joint model and factorizing it one way and so on.

Discriminative models, in some sense, do something that is more obvious. I know we've spent four weeks talking about these noisy channel models and doing things that way, and maybe it almost seems natural to you now. But discriminative models say that there's something that you've observed. I don't care what it is. It's just stuff you've observed. What I want to do is predict the stuff that I didn't observe, and I'm going to directly build a model of this conditional probability. So I'm not building any model of the probability of D.

I'm just directly modeling this conditional probability. So the difference for the MT case is that if we wanted to translate to French into English, we made a probably of E model and a probability of F given the E model, which meant that, implicitly, when you multiply those together by the chain rule, we have a joint model of the probability of E, F. Why did we do that? Well, there's this sort of halfway plausible story about how this lets you factor it into two problems, and each one is easier, and the language model could be reused, and there's a sort of a story there which is halfway believable.

But you could just think, “Oh, gosh. Why don't we just make a probability model which is the probability of English given French?” That's what we want, to translate from French into English. So that's what discriminative models do. So that then leads you into a space where there are methods like logistic regression, various kinds of exponential log linear sequence models, and also it's into the space of non-probabilistic models, like SVMs and perceptrons.

So this is what you get if you have kind of the base net picture of things. I'm not sure I should spend a lot of time on this. It probably splits between the people who have already seen this a lot, and therefore I don't need to review it, and the people who haven't it at all, and therefore sticking it up won't be very helpful for the. But the way that a lot of people think about these models these days is by drawing these kinds of arrow diagrams where you have the loads of the variables which take values, and the arrows are then showing the probability distributions that you're making.

So in the logistic progression model, you're just directly saying, “What's the probability of C, given these data things, where as for the generative model you do this funny reverse story?” So why would we be interested in doing conditional models? So answer one is – even if you changed nothing, in general, conditional models do perform better. So you really can just sort of take a system that's exactly the same, if it actually has a knob on it, and switch it from being generative to discriminative, and it will normally go up a bit in performance. So here's an example that does show this.

So this was a classification task where it was classifying senses of a word. So whether Jaguar is the operating system or the big cat – and so just for this experiment it was very carefully built, so it has exactly the same features that we use for prediction. There was exactly the same smoothing that was done by having pseudo-data. Everything was the same, and you could compare a generative model, which was naive based, and a discriminative model, which was a logistic progression.

Here's what you observe. If you look at the training set accuracy, the training set accuracy goes up massively, which should worry you a bit because it shows that discriminative models can really, really easily over fit data, which is something that we can talk more about later. But if we go to the test set, which is something that we should be looking at, we see that the performance has actually gone up by two and a half percent. So that's not so bad. You can just flip what you're doing from the generative to a discriminative model, and have your accuracy go up a nice lot.

So that's kind of a nice result. You might think that the reason why that is is because in a discriminative model, you're kind of making life easier for yourself. You're only spending your model resources trying to capture exactly what you're interested in, so all you want to do in this word sense task is to predict the sense of a word in context. So if you only set yourself that goal, it's easier to do well on it. Where as if you're building a joint model, that your joint model is actually trying to predict the distribution over all of the observed data. Things like how often different senses of word occur and what context occur around different senses.

It's actually trying to model the entire data set, and so that means you're wasting a lot of your modeling resources on something that you aren't actually interested in evaluating. So how are we going to build these models? The way that

we're going to build these models in a lot of modern discriminative NOP is done is in terms of having features, and then we're gonna put these features into classifiers. So what is a feature? Here's the confusing slide that says what a feature is. Maybe I'll show the examples first.

So here, the task that I am trying to do is part of speech taking. So I have to aide – and I'm looking at the second word here in blue, and maybe it could be a verb, or maybe it could be a noun. So when I define features, I'm defining predicates over some little bit of my data, and my predicate is going to form a function. It's going to be a function of some little piece of the data that will return a value. So here are some functions. So function one year says, "If defined class is noun, and the word I'm classifying is lower case and it ends with D, then this is a Boolean function. It'll return true."

So this purple feature is true of this data point where I'm making the classification decision of the last word. It's also true of this data point because the class is N, it ends in D, and it's lower case. But the purple function is false of these two data points. Okay, and my second classification function is that the class of noun previous word is to and previous part of speech was capital to. So that function is true of this sate of point, but it's false of every other data point, including this one here, this here. The class is verb.

So this is then yet a third one. So the idea here is that what I'll be able to do is actually kind of something like writing regular expressions; that you kind of can write these tricky patterns of regular expressions to match different things. Here, we're going to do the same thing abstractly, but we're going to call them features. So the way we'll be able to do all good NOP systems is that we'll think of any useful facts about configurations that will help us predict part of speech, right? A useful fact to predict part of speech is that if the proceeding word is a 'to', it's sort of likely the word after it will be a verb.

Not necessarily the case, but all else being equal, a word is more likely to be a verb if it follows the word 'to'. So we take any kind of observations like that that we think are sort of predictive of the sort of classes that we're interested in, and we call them a feature. So we write them down as a little function which will evaluate to a value, and that's our feature.

Student:[Inaudible]

Instructor (Christopher Manning):No, I'll say a bit about that in a moment, but no, they don't. But if I say that – something that is important to note, and in practice is often very useful is that it's good to have not only features that are kind of indicative and are good patterns, but also to have features that indicate something is wrong. So it's useful to have features like the word is 'the', and you've tagged it as a verb. That's highly likely to be a mistake, and so having those kind of error features that would match if you make a mistake is usually as or more useful than having features that match if you've done something correctly.

And often they're kind of easier to define, as well, in tricky cases. If you've got kind of a half way okay model that is still making some obvious mistakes, often the easiest way to address that is to look at how you can define some function that would match the kinds of errors the system is still making that wouldn't match good examples where it gets things correct, and adding those features into a model will then enable you to make it better. Okay, so this thing goes back to what are features? So crucially, features in general are a function that is a function of a piece of data, and a function of the class that is assigned to that data.

So they're looking jointly at both things. So an arbitrary feature can be of this form. That you're taking – you're writing a function on the data and the classes, and you're returning a real number. So no, they don't have to be binary or anything like that. In some context this is very useful, and certainly you want numeric value features of various times so that you might imagine in various kinds of web applications knowing the size of a picture that you might just measure in bytes. It might just be a useful feature, so in general you have arbitrary features. But it turns out that for a huge space of the NOP problems that we work on that there isn't so much need for the features that have real values.

It's partly a simplification, and it's partly other things. But in practice, if you look on the ground on what people do, the most common thing you see in practice and everything I present in these slides is that all of the features are actually predicates that have a binary answer. We're writing down some Boolean expression, and we're saying, "Is it true or not?" And so the features values are either one or zero. Even more than that, normally the style of features that you see is that the feature asks some question about the data, and then just says, "And the class is now, and the class is verb."

So the interesting part of the feature is the find of the data, and then you've just got an equality check on the class. So a lot of the time when people are talking about features that they really only explain what they're asking about the data, and then the assumption is that you then created features from this predicate which is then saying that the class is X. And you've produced X different features depending on how many classes that you've classified over.

Student:[Inaudible]

Instructor (Christopher Manning):Yeah.

Student:[Inaudible]

Instructor (Christopher Manning):Right, so we want to find the class, but the idea is that we don't know the thing in blue. Well, a way to address not knowing the thing in blue, and essentially, what we do is let's try out every possible candidate. We know the set of classes, and then as soon as we try out a possible candidate, and we say, “Okay, maybe it's VB.” we can evaluate a picture this and say that looks good according to this feature, or that looks bad according to this feature.

Although this is, first and foremost – having just these kind of features is sort of natural for a lot of natural language applications, like doing text classification, or part of speech tagging, or name data recognition, or word segmentation. Really, these are all the natural kinds of features. It's also worth noting the restricting of features to this form actually makes things more computationally efficient because since everything here has a zero-one value, that means that when you're actually implementing things, you can simplify the implementation because you're just sort of adding the active features rather than having to do all extra modifications and stuff.

Okay, right. So we have these features that we can then apply to data, and the idea is to each feature we're going to give a weight. The features weight is going be it's vote. It's going to say this is good, or this is bad. So our features here are saying that if we're assigning the class noun, and the proceeding word is two, which is part of speech tag two, that's sort of possible. So we're going to give it some weight as an okay thing, so maybe we'd give it a weight of .3. But really it's the proceeding word as to – it's more likely that it's going to be a verb, so we're going to give that a higher weight, .7 or something like that.

We're going to use the weights of the features as voting for a particular class, and so what we're going to do is look through all of our features that match in a certain configuration. So features are then kind of votes for and against a certain class assignment, and so we're going to tally up those votes, and we're going to decide which is the most likely class assignment. Okay, so features assign a weight so that when we're building our models to optimize classification accuracy. To do that, we're going to use these two quantities.

Firstly, any feature has an empirical count on our training data. So assuming hand classified training data to build the model off, and the empirical count of a feature is where we just take all of our data instances and try out our feature, and see if it matches. And we get an empirical count for the feature. We get that this feature matched 13 times for the data. I'm falling into the language here of assuming that all of my features are binary, but you can generalize them on binary count and still work out the expectation.

And then what we're going to want to do is build a model. The model also assigns some expectation to a feature in the usual way of expectations, so the model is going to be working out how likely different features are. So we have a model expectation for a feature. Okay, so I'm going to come back fairly soon on how to build these models. Let me give a few examples showing the kind of places and how these kind of models are used. So introducing the ideas of features and classes.

So very simple cases – text categorization, which is equivalent to logistic regression in a simple case. So here, our features might just be particular words, and the class that we want to assign is one of the number of text classes. So business, sports, world news, etc. If we do word sense disambiguation, we effectively do that kind of like text classification. We define a mini-document around the word of interest, just like the change of address example. We then have a context like this, and we then ask questions about that context. The questions might, again, be bag of word questions of is the word restructure nearby.

But then we can also ask more precise questions, so we can ask questions like, “Is the next word 'debt'?” So we have richer classes of features that we can ask about. The more interesting case that we'll work towards is doing the typical kind of problems which commonly get called sequence problems. The way that we're going to handle sequence problems is we're going to walk through the text, and we're going to handle the sequence problems one position at a time. So at any particular point, we're going to build a classifier, assuming that we've already done everything up to a certain point, so our picture looks like this.

What we're going to be wanting to do is build a probabilistic model over which part of speech we should put above the word fall. So the label is the choice of a part of speech here now, and the kind of questions we can ask is, “What's this word? What's the previous word? What's the previous part of speech?” We can also ask about what is the next word because all of the data is assumed to be observed. The things that we can't ask are about following classes because we haven't yet figured them out yet.

So here are just a couple of slides giving a few results on task and motivating some of the properties of this. So this is simple text classification resolve, which is a paper from Jungian Oles. I guess Jungian was a Stanford SCCM graduate. So they're doing text classification over this Reuter's data set of news articles, which is a very classic text classification data set. They basically compare a number of different classifications, algorithms, and here are their results. So naive based is a kind of a classical text classification base line, and it's what everyone presents as their base line for text classification.

On this data set it's getting 77 percent F1. We remember from last time, the F1 is doing this harmonic mean between precision and recoil at the class level. Honest people like to report text categorization results using this F1 measure for the classic information retrieval reason. A lot of the time in those machine-learning papers they don't do that, and you just see accuracy results for text classification. But those results tend to be extremely misleading because typically you have a very skewed distribution of classes and text classification problems. You can get high accuracy results, even if you always or almost always get lots of the small classes wrong.

So doing F1 evaluation is a much more rigorous measure as to whether you're actually doing a decent job at getting rare classes right. So naive base here got 77 percent F1. Not terrible, not great. Here are three other systems, and they all get about the same, as you can see. So something that is sort of interesting – so all of these are using bag of words features of what are the words in the document, and throwing them into building a classifier. The thing that's sort of a little bit interesting here is that any statistician would tell you that doing a linear regression over this categorical count data of word occurrences and documents is that all of the assumptions of the model are wrong. You shouldn't be doing it.

Where as in some sense the assumptions of the naive based model are right. It is the right kind of categorical model to be using for building text classification problems, but the result that you actually seem to get is if you build a discriminative model, it works much better, and it kind of almost doesn't matter which one you choose. It doesn't even matter if it's one that is supposedly highly inappropriate in terms of it's assumptions. Then all of them work a lot better than the generative model. One of the things that they talk about again and that I'll get back to is this question of how to avoid discriminative models over-fitting.

Maybe I've sort of said this by this point because I kind of went through an example of the part of speech taking, but this is perhaps more clearly illustrating how we're up to a particular point in making our decisions in the kind of features we can use, and what's the current word, what's the next word, what's the previous tag? You can also ask about joint features and commonly do, so what's the sequence of two previous tags might be useful, joint feature. Commonly we can define a lot of the main specific interesting and useful features.

So while asking about the word is useful, it has this problem that words are very sparse, and a lot of the time you won't have seen words. You won't have seen them in a particular context. So a lot of the time you can get a ton of value by defining your own equivalence class features, like does the word have a digit in it? In some sense you could think of this as an extremely large disjunction over words, right? Is the word two or is the word three, or is it four, or five, or 22.6? In practice, using hand-designed functions that pick out the main, useful classes of data items is just a highly useful technique in improving these kinds of models in practice.

It's one that is typically under-exploited by people. People who build NOP systems and get them to work well spend a lot of the time thinking about how they can hand-define good features that will pick out things just right. Where as if you just throw in the most obvious features of what's the word and what's the tag in practice, then the systems sort of work, but not so well. So this kind of discriminative modeling, you can kind of apply it to any other kind of problem as well in NOP, and essentially people have.

Sentence boundary detection, what do prepositional phrases modify, lots of different things. I think I've sort of said that, except for the bottom bit. Okay, so I've sort of said twice what joint and conditional models are. The next important thing to know about them is how easy are these models to make? Not withstanding if you're having some troubles with assignment two at the moment. In principle, doing joint models over categorical data is really easy, and it's really fast to build models.

That's because it's a theorem that optimizing joint likelihood over categorical models is just using relative frequencies. You count and you divide, and you've got the next in a likelihood model. Conditional models turn out to be much harder to do because optimizing conditional likelihood – you can't just count and divide, so you have to do interactive methods of numerical optimization, and I'll get onto that.

So these kind of ways in which we are building features and collecting votes brings us into the space of what are called linear classifiers. A linear classifier is kind of a useful notion to have because most of the machine learning methods that people use most of the time are different forms of linear classifier. So the picture for a linear classifier is you have features. You want to assign some class. Each feature has a weight, and the way you work out which class to choose is you work out a vote for each class, and the vote for each class is you see each feature's match on a particular choice of class in the observed data.

Then you take the weight of that feature, and you just sum up these votes. If you have binary features then it's literally as easy as this. For 2A, it can be tagged as either a noun or a verb. Let's say that there are only those two choices. For the features I had before, these two features match it being noun tagged. This feature matches it being verb tagged. So if these features have the weights shown, the vote for it being tagged as a noun is minus .6 because I just summed those two numbers. The vote for it being tagged as a verb is .3. .3 is bigger than .6, and therefore we tag it as a verb. That's what a linear classifier does.

All linear classifiers do that. It's not actually such an advanced thing to do, really. So all of the trickiness beyond that comes down to how exactly do you choose these weights. There are a bunch of algorithms that choose the weights in different ways. One of the simplest algorithms of linear classifiers is perception algorithms, which effectively do this on-line learning to nudge weights around until you stop making mistakes.

SVMs do a linear classifiers and the basic case. They do a different kind of optimization. What we're going to do here for exponential models, which have a whole bunch of names, log linear, maxent, logistic, gibbs, etc. models. What we're using to set the weights is that we're going to set the weights so we maximize the conditional likelihood of this exponential model form. So we've chosen this particular exponential model form, and then we're going to say, “Okay, they want to fiddle these landers so that we give the highest conditional likelihood possible toward the observed data.”

Okay, what is this model form here? One way of thinking of it is sort of the following crude way. We want to be building a probabilistic model, and what we have is features with weights like this. The first problem we have is that these things can go negative because if you sum these two, minus .6, that's bad for a probabilistic model. So one way to solve that is we exponentiate because if we exponentiate, everything is positive. That sounds good for a probabilistic model. Then for probabilistic models you want to normalize and turn into a probability distribution. The way you do that is by considering different classes, and working out the same quantity for them, and then just dividing through.

And behold, you have a probability distribution in two easy steps. So in some sense you can think of it as no more complex than that. There is a lot more motivation for why this exponential model form is a natural and good form for modeling these kinds of decisions, but I won't get into that right now. Once we have this model form and features that give votes, we can then get out a conditional probability over class assignments.

So using my previous example and the same active features that we had – the two features weighed 1.2 and minus 1.8, voting for noun. So we take the sum of their votes and exponentiate it. E to the minus .6, and then for it being a verb, we're taking the vote for that. E to the 0.3. Then for both of them, we're dividing through by the same normalization term, which is just some of this fuss and some of that. We do our little bit of math, and we get out that this model is saying that it's a 29 percent chance that a noun should be assigned, and a 71 percent chance that a verb should be assigned.

Does that make sense up to there? Okay, right. So that's the model class that I'm going to talk about and that we're going to go through the math of, and you're going to use for the assignments. It's a very standard, commonly used, very good way of doing things about it. It's absolutely not the only way to do it. For instance, another model class that there's been a lot of work on it in machine-learning is using support vector machines, which – setting those weights in a different way to optimize a different criteria.

Another good way to do things – I'm not gonna do them in this class. Okay, something that is worth noting is that for my picture here of saying that this is our exponential model. We've got features, and we're putting weights on the features. Actually, a naive based classifier is actually an instance of an exponential model. So if the naive based classifier – the probability of a class, given a data, was the prior probability of the class, and then the product of the probability of each observed word, if it's something like a unigram model, given the class. Then again, we'd have a normalization turn where we consider all different classes.

So if you do a little bit of math on that, in particular you put in these ex-blog pairs, and you assume that your feature's log probabilities – that what you get out is that that model just becomes the same exponential form where your feature is the word dishwasher and the document corresponding to the probability of dishwasher, given the class. Then your weights are the log probabilities. So naive based is just an exponential model. The question of how do they differ comes down to the question of you're choosing to set Thelma Parameters. In a naive based model, you are setting Thelma Parameters to maximize the joint likelihood by setting them to the observe conditional probabilities in the data.

From what we can do now for discriminative classifiers, we're instead gonna set these Thelma Parameters so as to maximize the likelihood of the decisions that we're gonna make. So that leads into these next couple of examples in which I hope not to confuse you, but instead to show you these exciteful, wonderful examples that these things are different and do deliver different results. So what we have here is we are going to be sensing whether it's sunny or raining. We're assuming that we kind of have a not very noisy sensor outside our window which attempts to sense whether it's sunny or raining.

It's not very good. So the sensors make mistakes. In fact, they only get things right 3/4th of the time. So if it's raining, it's going to say plus three quarters of the time, and if it's sunny it's going to say minus 3/4th of the time. So that's what the sensor is like. But then for this example I have – there's an extra bit of art in here. Let's suppose I actually only have one sensor outside my window, but there are two wires running from it, and I wrongly think that I've actually got two sensors outside my window, but they're both wired together and actually report the same thing always.

So the configurations I can then get is that if the sensors are – my two wires, when I measure them with my little whatever, are both going to be saying the same thing. So if it's raining outside, the two things I can get is plus-plus, which I will get three quarters of the time it's raining. And if I soon hear that overall it's raining half the time – this is assuming we're in Seattle or something, then 3/8th of the time this is my observation. Similarly, a quarter of the time when it's raining, they don't detect that and they return minus, so that data, overall, happens 1/8 of the time, and then the reverse over here.

So my naive based model factors are half the time it's sunny, and if it's sunny, it'll report plus a quarter of the time and minus three-quarters of the time. If it's raining, it'll report plus three-quarters of the time, and minus one-quarter of the time. Then what I want to do is work out what probabilities my naive based model predicts, given that I see plus-plus. If I see plus-plus, look at this picture. It's obvious what it should predict. In the cases where we're getting plus plus, it should be predicting that that means that three-quarters of the time it's raining, and one-quarter of the time it's sunny. That just is the observed data. All right?

It should report that the probability of rain, given plus-plus, is three-quarters. But what I'm doing is measuring the sensor twice, and I'm assuming that my two wires are independent measures where they aren't really. Because I'm making that wrong independence assumption, my probabilities come out all wrong. So the joint probability of plus plus rain is the prior probability of it being raining a half times the probability of plus given rain is three-quarters times three-quarters. And I work out the other cases, and I actually end up way over-confident. The probability of raining, given that I see plus plus, is nine-tenths according to a naive based model, and it's only one-tenth chance of being sunny.

That make sense? Okay. So that's – now, you might say that this is a very artificial example, but what I'm going to want to argue is that it's not an artificial example. Yes, this is an artificial example, but the problem with illustrating isn't an artificial example.

It's exactly how things are in NOP all the time, right? So the problem is that if you have overlapping evidence, so you're measuring things multiple times that aren't actually independent of each other, there is this systematic problem with generative models, like the naive based model, that – naive based models multi-count the evidence.

If you're trying to measure things that are correlated or overlapping or something, it will treat it each times independent, and it will count the evidence many times. So crucially, what maxent models will do is that they'll solve that problem. I'll get back to it in more detail later. A maxent model will work out how to adjust the [inaudible] weight so it's not multi-counting evidence. So that – all I showed so far was an example where the naive based probability estimates were wrong. And you could just think, "Well, so what. The probability estimates weren't that good."

But does that cause problems in practice? So here's a further example which is kind of cooked up in the same way. It illustrates how making these false independence assumptions does actually cause you to make the wrong classification decisions. So this time we're at an intersection, and we can see two traffic lights. We can see the traffic lights on these two sides. So there are two possibilities – sorry, there are three possibilities. The traffic lights could be working, and this one could be green, and this one could be red. Or they could be working, and this one is red, and this one is green. Or the traffic lights could be broken, and they're both showing red.

Let's assume that this is the real data distribution so that 6/7ths of the time the traffic lights are working, and when that's true they flip back-and-forth. So half the time each, I think 3/7th of the time each you're getting this configuration and that configuration. Then 1/7th of the time the lights are broke, and you see this configuration. Okay, so what we have here is this light and this light. Their behavior is correlated. But let's assume we were building a naive based model, and that we treat those two lights as separate pieces of evidence.

So then we have a prior probability of the lights working, and we have the probability of each light being green or red, conditioned on whether the lights were working or not. So the probability of working at 6/7th, and then given it's working, you're equally likely for each light to be green or red, so you get these probability factors. The probability of broken is 1/7th, and then if it's broken with probability one, you're seeing a red light.

Okay, and now here is where things go broken. If we build that naive based model and we show it two red lights, what does it do? What it should do is look up here. If both lights are red, you should conclude that the lights are broken. But that's not what a naive based model that models the two lights separately actually does. So what we find is that the probability of broken red-red is the probability of broken is 1/7th times the probability of red given broken is one times the probability of red given broken is one equals 1/7th. Where as the probability of working in red-red is 6/7th prior probability of working than a half chance of seeing red given working times a half, which equals 6/28th.

If you do your math, this number is bigger than that number, and so what the model actually predicts is the probability of working, given that you see red-red is 6/10th. So therefore the classifier will return working when you see red-red. That's bad. Because the naive based classifier was sort of set from these generative relative likelihood probabilities – that, you know, it is a theorem that these probabilities – the ones that optimize the likelihood of the observed data within this model, but that actually causes it to make the wrong classification decisions.

So the idea that motivates discriminative models working better is why do we have to choose the relative frequencies if all we want to do is to get classification decisions to be right? Suppose we change the numbers. Could we then get the things to work out right? We could. Suppose I just said – suppose I just changed this one factor because I felt like I had said, “Let me change, in my model, the probability of broken to be a half.” You know, that's not true in the observed data. That wasn't the right answer. Suppose I just did it anyway. Then I'd get the probability of broken red-red being a half, and the probability of working red-red being an eighth because now I've just changed these numbers.

So now I predict that if I saw red-red that the lights were broken because there's only a 1/5 chance of them working. So by fiddling the numbers, I can cause a classifier to make the right discriminative conditional classification decisions because I've stopped saying I have to be modeling the joint likelihood of the observed data. So in precisely that way you can improve your classification decisions by not staying true to the observed data likelihoods.

Student:[Inaudible]

Instructor (Christopher Manning):No, you can actually work through it. If you make this change, the model now predicts correctly in each case. If you give a red-green, it predicts – I mean, it's kind of obvious. Since this is zero, if you've given an input of red-green or green-red, it says with probability one that it's working. Now if you give it red-red, it says with probability 0.8, it's not working. So this is sort of just a summary of where we're up to. So to build a maxent model, we do effectively linguistic data modeling, and we define what are good features.

Then each of the features that we're gonna want to give them a weight. We're gonna want to give them good weights. So to give them good weights, what we're gonna use is the count of the feature on the training data. That's empirical expectation. Then we're going to want to work out a model expectation so we can work out the data's conditional likelihood according to the model. What we're then going to want to do is fiddle those weights with no other goal than to improve the conditional likelihood of the decisions, given the data. That quantity at least seems closely related to the classification decisions that we want to make, and so we hope it'll work well.

So the next major part of this, which I'll get into as far as I can today, is how do we go about setting the weights? The way we go about setting the weights is we have to do some math. What we want to do is we want to set these landers so as to make the conditional likelihood of the classification decisions as high as possible given the data that we have. Then, as is usual, we actually maximize the log-conditional likelihood more than the likelihood because you have more summations that way, and so life is easier.

So that's the log of the product over the individual classification decisions in our whole data set, which is the sum of the log of the individual classification decisions in our whole data set. So this quantity here – well, we know what that is. It's just the log of this exponential model form that we wrote out before. We can calculate this for any particular feature weights, and any functions, and any data sets. Is this easy to calculate? In practice whether this is easy to calculate or not depends on how many possible classes there are to assign.

It's fairly easy to calculate this because you are assuming a particular observed path and observed data set. This is the supervised training data, and you work out which features apply. Sum them up and exponentiate. The thing that is potentially a killer is in the nominator you have to consider here summing over all possible class assignments. So in practice of the number of possible classes is small, like if you're classifying e-mail messages or spam, you haven't got a problem. But if the number of potential classes is extremely large, you've got an enormous problem.

That's one of the reasons why to this day, almost invariably, language models are built as generative models because if you start thinking, “Well, can I use this technology we're learning about to build a discriminative language model?” Well, the problem is that in a language model, you want to predict what word comes next, and there's a huge space of possible words. If you have higher order features then there's an even huger space of bi-grams and tri-grams. So then trying to calculate this becomes completely infeasible suddenly by exact inference. Yes?

Student:[Inaudible]

Instructor (Christopher Manning):So it turns out that's not a problem, in general. Like everything, like all of these things, we can buy our machines with lots of memory and so on. But in the typical case, that's not actually such a bad problem, and that comes back to when I said that you make it a lot easier for yourself, computationally, if you have only binary features. Because the typical situation that you have is that the total space of features is very large. So a lot of the time and good NOP discriminative classifiers – the total space of features is that you might have 5 million of them, or you might have more.

But merely – most of those features are very specific to particular context, so in a particular context, the number of features that actually match is typically very small and might only be 20 or 30 because most of the features saying things like the word is circus, or the previous word is housing, and the word before that is cheap. There are very, very specific features, and normally almost none of them match a particular situation. So there are very few that match.

That means that you can do this sufficiently, providing you can find which features match in a particular context, and that's where you can then start into your computer science tricks of doing clever indexing so you can very efficiently find the features that match. Where as there kind of isn't any solution for that here because if you're doing it over all classes – well, you're doing it over all classes.

So we went to fiddle the lanthas to maximize this log-conditional likelihood. How can we do that? Well, we can separate – since this is a log of, whatever, a division thing. We can separate it into two parts. We can have the numerator part minus the denominator part. The numerator part is just considering our observed data. So how are we going to maximize the lanthas? At this point, it's a numeric optimization problem. We want to set those lanthas so this number gets as big as possible.

So that means that we do differentiation of this, and say, "Okay, let's differentiate this and see if we can get this." Well, let's differentiate this and see if we can find the maximum, and the maximum will be where this quantity equals that quantity. The derivative of this quantity equals the derivative of this quantity, so you're a maximum of the function. I almost said that wrong. Yes?

Student:[Inaudible]

Instructor (Christopher Manning):I'm trying to max – no. I'm trying to maximize it for my supervised training data, so I'm working out this quantity, which is all of my supervised training data. So I'd have some big set of data, capital CD, consisting of individual data points with classes assigned to them, and that's the outside summation here is over all of my data points. But for all of them I have observed answers.

Okay, so I do differentiation. So doing the numerator is really, really easy because you have to log in the X next to each other so they just go away, and then you have a partial derivative. So you're working out partial derivatives with respect to each lantha eye, and each weight in the model. So the partial derivative with respect to lantha eye goes to this summation, so you can move the partial derivatives inside the summation.

And then, since you're doing the partial derivative with respect to lantha eye, the only thing that remains is what is the thing that lantha eye is multiplied by, which is this feature: FI of CD. Summed over all of your data set. So the derivative of the numerator is this empirical count. How many times did the feature match in the observed data? The derivative of the denominator is vaguely more tricky because you have to remember the chain rule. I guess that's why they have these math requirements in engineering.

So we want to differentiate the denominator term, and so then we have to have – so we want to differentiate this log of something, so we've used the chain rule. So we'd have the derivative of the log is the inverse, and so you have the one on the rest of it times the derivative of the inside there. And then you do the chain rule a second time, and you have the derivative of X is X. So you have the X of the rest of that times the derivative of the inside. Then for this derivative of the inside bit here, that's the same as before that the partial derivative, with respect to lantha eye, everything goes away, apart from FI of the class down here.

Then low and behold, if you stick this bit over this bit, you get back exactly the model form that we had for the exponential model. So what you end up with is that you are summing over all of the data points in your training data, but then you are summing for every possible choice of the class decision at each point. The probability that the model assigns to that class decision times the feature, which in our case is just this one zero decision. So this is saying, “Well, what's the model expectation? The models predicted number of times that this feature should match according to the data?”

So what you're actually doing – all right. So you're derivative is how many times the feature actually matches on the data minus how many times the model predicts the feature should match on the observed data that you've given it. So therefore to maximize the conditional log likelihood, you want the derivative to be zero, and so therefore you're wanting to make the actual count of a feature equal to the predicted count.

So in the optimization that you're actually doing, you've measured the empirical counts of every feature on your training data, and then you're trying to fiddle around the lantha weights so the model expectation, the model saying, “Okay, according to my model, this feature should be firing 7.2 times. It's actually firing six times on the observed data, so I have to fiddle down the weight of a feature so that I'm predicting it fires less times.” So that's what we want to do.

Theorem – this is actually a nice problem. It's convex, so you can find an optimal answer, which is the best weighting of parameters. It can be under constrains, so actual weights might not be unique, but there is a maximum to the conditional likelihood that you're assigning. Here's a neat property that – the maximum always exists as this, providing you play nicely. Why don't you just count true feature counts of observed data. You have to have a well formulated problem that has a maximum.

If you start trying to play tricks, and say, “I don't like what it's doing, so why don't I just change a few of the empirical counts that I observed in the data.” then you can easily make your model constraints inconsistent, and then there'll be no solution. So I presented these as effectively exponential models, and in terms of what I've presented, they've just been exponential models. Despite that, in NOP these are always referred to as maximum entropy models, but I think I'll leave it until next time in sort of saying why those also get called maximum entropy models.

So at the end of the day, what we have is these weights. We want to set them so that we maximize the conditional log-likelihood of the training data. At this point, the conditional log-likelihood is a convex problem. You can set the lanthas, you can find the maximum. You can do it with any numerical optimization method, and there are just then questions as to which numerical optimization method you use. In traditional work in statistics and NOP, people tended to use these generalized sort of scaling methods, or proportional fitting style methods.

Although I still have them on the two slides there, they are basically abandoned these days because everyone has found that you can do things much more efficiently using more general numeric optimization methods. If you've done that kind of stuff, you can think of things like conjugate gradient methods, or Quasi-Newton methods. If you haven't done that kind of stuff, don't worry about it. We're gonna give you an optimization package, which is a limited memory Quasi. Use an optimizer, and these kinds of optimizers are essentially the state of the art for doing optimization of large scale discriminative learners of this size type.

Where in particular the trick is all in this limited memory bit, so the Quasi-Newton methods, in general, construct the hechen, which then you've got this matrix of partial derivatives. And the problem is that that's completely untenable if you have 5 million features. So all of the trick of limited memory runs isn't there, trying to approximate that while using a much more reasonable amount of storage.

Okay, I think it sort of makes sense for me to stop here today, since the next bunch of slides go into the motivation from the angle of maximum entropy, and so it doesn't make sense for me to have done one of those slides and then stop. So I'll call that enough for today, and then move on to the second half next time.

[End of Audio]

Duration: 74 minutes